Entwicklungsrichtlinien für die Programmiersprache JAVA

Berner, Stefan ; Joos, Stefan ; Glinz, Martin

Abstract: Entwicklungsrichtlinien sind ein Hilfsmittel, um gemachte Erfahrungen bei der Entwicklung von Software weiterzugeben. Sie helfen Entwicklern, vorhandenen Programmcode zu verstehen und in zukünftig zu erstellendem Programmcode Fehler zu vermeiden. Konsequent und umsichtig angewandt, verbessern sie den Programmierstil und die Lesbarkeit von Programmcode und tragen somit auch zu verbesserter Wartbarkeit von Software bei. Wie Entwicklungsrichtlinien für die Programmiersprache Java sinnvollerweise aussehen können, was sie enthalten sollten, wie sie gegliedert werden, wie man sie anwendet und aktualisiert, ist Inhalt dieses Beitrags.

Originally published at:
Berner, Stefan; Joos, Stefan; Glinz, Martin (1997). Entwicklungsrichtlinien für die Programmiersprache JAVA. Informatik: Zeitschrift der schweizerischen Informatikorganisationen, 4(3):8-11.

Entwicklungsrichtlinien für die Programmiersprache Java

Stefan Berner, Stefan Joos, Martin Glinz
Institut für Informatik der Universität Zürich
{berner, sjoos, glinz}@ifi.unizh.ch

Entwicklungsrichtlinien sind ein Hilfsmittel, um gemachte Erfahrungen bei der Entwicklung von Software weiterzugeben. Sie helfen Entwicklern, vorhandenen Programmcode zu verstehen und in zukünftig zu erstellendem Programmcode Fehler zu vermeiden. Konsequent und umsichtig angewandt, verbessern sie den Programmierstil und die Lesbarkeit von Programmcode und tragen somit auch zu verbesserter Wartbarkeit von Software bei.

Wie Entwicklungsrichtlinien für die Programmiersprache Java sinnvollerweise aussehen können, was sie enthalten sollten, wie sie gegliedert werden, wie man sie anwendet und aktualisiert, ist Inhalt dieses Beitrags.

1 Einleitung

Mit der Sprache Java als Teil der Java-Plattform ist eine neue objektorientierte Programmiersprache entstanden, bei deren Entwurf viele moderne Sprachkonzepte berücksichtigt wurden, zum Beispiel eine strenge Typbindung, die Möglichkeit zur separaten und eigenständigen Definition von Schnittstellen, die Behandlung von Ausnahmesituationen mittels *Exceptions* und vieles mehr. Wenn man von der stark an C angelehnten Syntax der Sprache einmal absieht, dann erleichtert die Sprachkonzeption an sich bereits die Erstellung von robustem und wartungsfreundlichem Programmcode (kurz Code). Leider lassen sich nicht alle Aspekte diesbezüglich über die Sprachdefinition regeln. Dies liegt hauptsächlich daran, dass ab einer gewissen Komplexität die Sprache unhandlich und somit wieder fehleranfällig würde. Ausserdem gibt es Aspekte, die nicht über die Sprachdefinition zu regeln sind, wie beispielsweise die Bezeichnerwahl oder das »sich Zurechtfinden« in fremdem Code. Mit wachsender Erfahrung in der Entwicklung von Software und im Umgang mit der Sprache lernt ein Entwickler im allgemeinen neben den sprachunabhängigen Problemen auch die »Haken und Ösen« der Programmiersprache kennen und zunehmend beherrschen. Dieser Lernprozess ist oft mühsam und langwierig. Eine Möglichkeit, ihn zu verkürzen, besteht darin, gemachte Erfahrungen zu sammeln, aufzubereiten und in Form von Entwicklungsrichtlinien bereitzustellen.

Die Entwicklung von Software ist ein kreativer Prozess mit vielen potentiellen und gangbaren Lösungsmöglichkeiten. Richtlinien sind daher eine heikle Angelegenheit, da sie per se dazu da sind, einen einheitlichen Stil vorzuschreiben. Entwicklungsrichtlinien können nicht und sollten sicherlich auch nicht versuchen, *den* einzig guten Programmierstil zu definieren. Wohl aber können sie einen zweckmässigen, einheitlichen Stil vorgeben. Ferner können sie bei Software-Inspektionen als Grundlage für Prüf- und Abnahmekriterien dienen.

Konsequent und umsichtig angewandt, sind Entwicklungsrichtlinien ein wichtiges Hilfsmittel, um Arbeitsweisen zu vereinheitlichen, gemachte Erfahrungen weiterzugeben, die Dokumentation der Software zu verbessern und sich in eigenem wie in fremdem Programmcode besser zurechtzufinden. Damit tragen Entwicklungsrichtlinien

Stefan Berner studierte Informatik an der Universität Stuttgart. Seit Ende 1994 arbeitet er als Assistent und Doktorand in der Forschungsgruppe Requirements Engineering am Institut für Informatik der Universität Zürich. Er beschäftigt sich dort mit teilformalen Spezifikationsmethoden. Schwerpunkt ist hierbei die Visualisierung von Spezifikationsmodellen sowie Werkzeugkonzeption und -bau.
Stefan Joos studierte Informatik an der Universität Stuttgart. Seit Anfang 1994 arbeitet er als Assistent und Doktorand in der Forschungsgruppe Requirements Engineering am Institut für Informatik der Universität Zürich. Er beschäftigt sich dort ebenfalls mit teilformalen Spezifikationsmethoden. Schwerpunkt seiner Arbeit ist die Definition einer geeigneten Spezifikationssprache.
Martin Glinz studierte Mathematik und Informatik an der Technischen Hochschule Aachen, wo er 1983 zum Dr. rer. nat. promovierte. Danach beschäftigte er sich zehn Jahre bei BBC/ABB in Baden mit Forschung, Entwicklung, Schulung und Beratung im Gebiet Software Engineering. Seit 1993 ist er Professor für Informatik an der Universität Zürich. Er leitet dort die Forschungsgruppe Requirements Engineering.

auch zur Verbesserung der Wartbarkeit von Software bei.

Vor dem Hintergrund »Software-Entwicklung an der Universität«, d.h. Ausbildung, Betreuung studentischer Arbeiten und Software, die im Rahmen von Forschungsprojekten erstellt wird, haben wir »Entwicklungsrichtlinien für Software-Entwicklungen in der Programmiersprache Java« erstellt [Berner et al. 96]. Diese können auf Anfrage oder über WWW unter »http://www.ifi.unizh.ch/groups/req/publications/selected_publications.html« bezogen werden.

2 Inhaltsübersicht

Auf der Grundlage der Erfahrungen mit den Java-Entwicklungsrichtlinien unserer Forschungsgruppe beschreiben wir in diesem Beitrag, wie Entwicklungsrichtlinien für die Programmiersprache Java aussehen können, was sie enthalten sollten und wie sie gegliedert werden. Abschnitt 3 geht auf zwei unterschiedliche Möglichkeiten ein, wie Entwicklungsrichtlinien aufgebaut und gegliedert werden können. In Abschnitt 4 wird dargelegt und anhand von Beispielen illustriert, welche Bereiche Entwicklungsrichtlinien für Java regeln können und sollten. Abschnitt 5 behandelt die Anwendung und Änderung der Richtlinien.

3 Aufbau der Richtlinien

Der Aufbau und die Gliederung der Richtlinien trägt massgeblich zu deren Anwendbarkeit bei. Die Art der Gliederung sollte auf den intendierten Verwendungszweck abgestimmt sein. Sollen die Richtlinien hauptsächlich in Form eines Lehrbuchs genutzt werden, d.h. sorgfältig gelesen, verinnerlicht und anschliessend beiseite gelegt werden, so bietet sich eine Gliederung an, die sich an den *Sprachelementen* und der *Struktur der Programmiersprache* orientiert. Alles Wissenswerte über Anwendung, Dokumentation, Kommentierung, etc. bestimmter Sprachelemente ist jeweils dort zu finden, wo eben diese Sprachelemente beschrieben sind.

Wird in den Richtlinien eher eine Art von Arbeitsdokument oder Nachschlagewerk gesehen, dann ist es angebracht, die Richtlinien nach *stilistischen Themen* zu gliedern, z.B. Programmierhinweise/-stil, Einrückungen und Layout, Bezeichnerwahl, Dokumentation und Kommentierung, etc. Innerhalb dieser Kapitel kann entsprechend der Struktur der Programmiersprache auf die einzelnen Regelungen eingegangen werden. Beispielsweise wird im Kapitel »Dokumentieren von Java-Code« auf das Kommentieren von Paketen, Klassen, Methoden, etc. eingegangen, aber nicht auf Einrückung und Layout derselben. Dies wird in einem eigenen Kapitel »Einrückungen und Layout« behandelt.

Für unsere eigenen Java-Entwicklungsrichtlinien haben wir uns für letztere Art der Gliederung entschieden, da wir mit Smalltalk-Entwicklungsrichtlinien [Schneider94] die Erfahrung gemacht hatten, dass die Richtlinien oft in Form eines Nachschlagewerks genutzt werden. Typische Anwendungsszenarien waren überwiegend sachgebietsorientiert, etwa in der Art: Wie wird dokumentiert? Wie werden Methoden dokumentiert? Wie wird eine Vorbedin-

ABBILDUNG 1 Inhalt und Gliederung von Entwicklungsrichtlinien im Sinne eines Nachschlagewerks. Beispiel aus [Berner et al. 96].

gung angegeben? Wo finde ich alles Wissenswerte zum entsprechenden Thema? Fragestellungen wie beispielsweise »Wo und wie erfahre ich alles über Klassen?« waren selten.

Abbildung 1 zeigt die Gliederung unserer Entwicklungsrichtlinien für Java. Die Grobgliederung ist generisch, d.h. die Einteilung in Kapitel kann (weitgehend) unverändert verwendet werden, um ggf. weitere Richtlinien für andere Programmiersprachen zu gliedern. Da in Entwicklungsrichtlinien unter anderem auf Besonderheiten und Probleme der Programmiersprache eingegangen wird, ist die Aufteilung der Kapitel in Unterkapitel zumindest teilweise sprachabhängig und kann nicht unmittelbar wiederverwendet werden, um Richtlinien für andere Programmiersprachen zu gliedern. Bei der in hier gezeigten Gliederung gilt dies insbesondere für die Kapitel 2 und 3.

4 Inhalt der Richtlinien

Wesentlich ist unserer Meinung nach die Regelung der in Abbildung 1 aufgeführten Bereiche. Existieren bereits Richtlinien für Entwicklungen in anderen Programmiersprachen, so sollten bewährte Regelungen stets berücksichtigt werden, d.h. miteinfliessen oder ggf. entsprechend den Gegebenheiten der neuen Programmiersprache und Entwicklungsumgebung angepasst werden. Dies fördert die Akzeptanz der Richtlinien und trägt dazu bei, dass langfristig eine firmen- und problemangepasste Entwicklungskultur entsteht.

Für jede einzelne Regelung der Entwicklungsrichtlinien sollte sorgsam geprüft werden, ob sich Beispiele finden lassen, wo diese sinnvoll anzuwenden ist. Finden sich keine Beispiele, ist dies ein Hinweis darauf, dass eine Regelung überflüssig oder unpraktikabel ist. Aussagekräftige Beispiele sind auch ein wichtiges Hilfsmittel, wenn es darum geht, diejenigen Regelungen zu dokumentieren, deren Sinn und Zweck nicht unmittelbar ersichtlich ist.

Grob gesehen können Entwicklungsrichtlinien zwei Arten von Problemen angehen. Zur ersten Kategorie (1) gehören Probleme der Programmiersprache, die über die Sprachdefinition gelöst werden könnten, aber nicht gelöst worden sind. In diese Kategorie fällt in Java beispielsweise die Verwendung potentiell fehlerträchtiger Anweisungen wie der Switch-Anweisung, alle Arten von In- oder Dekrement-Anweisungen oder

… auch Blöcke, die nur aus einer Anweisung bestehen, *müssen* geklammert werden, auch wenn dies rein syntaktisch gesehen nicht notwendig wäre. Dadurch wird die *Dangling Else-Mehrdeutigkeit* der `if`-Anweisung umgangen und explizit gemacht, zu welcher `if`-Anweisung ein `else` gehört. Ferner wird der häufig gemachte Flüchtigkeitsfehler vermieden, die Klammerung zu vergessen, wenn nachträglich der Bedingungsrumpf erweitert wird …

ABBILDUNG 2 Eindeutiges Auflösen von Mehrdeutigkeiten der Sprachgrammatik und Vermeiden von Flüchtigkeitsfehlern. Beispiel für eine Regelung der Kategorie (1) aus dem Kapitel » Einrückungen und Lay-

… Java hat ausser Arrays und Klassen keine benutzerdefinierbaren Typen und somit auch keine Aufzählungstypen oder Bereichstypen. Dieser Mangel ist sicherlich nicht substantiell, da er durch die Verwendung von Klassen kompensiert werden kann, was jedoch aufwendiger ist. Das Vorhandensein von Aufzählungstypen in einer Programmiersprache erleichtert generell deren Verständlichkeit, da Ausdrücke wie in (a) abgebildet möglich sind. Sieht man von einer Verwendung von Klassen ab, dann würde ein äquivalentes Code-Stück in Java in etwa so aussehen wie in (b) abgebildet.

```
TYPE        // (a)
  TrafficLightType
    = ( red, green, yellow );

VAR
  trafficLight
    : TrafficLightType;

BEGIN
  …
  IF ( ampel = red ) THEN
    …
  END
  …
END
```

```
            // (b)
…
void foo()
  {
  /* 0=red, 1=yellow,
  2=green */
  int trafficLight;
  …
  if ( trafficLight == 0 )
    {
    …
    }
  …
  }
```

```
class TrafficLight// (c)
  {
  boolean isRed() { … };
  boolean isYellow() { … };
  boolean isGreen() { … };
  …
  void fooToo( … )
    {
    TrafficLight trafficLight
      = new Ampel();
    …
    if (ampel.isRed())
      {
      …
      }
    …
    }
  }
```

… Das Code-Fragment (b) hat im Vergleich zu (a) den Nachteil, dass es weniger gut verständlich ist, insbesondere wenn Variablendeklaration und -verwendung weit auseinander liegen. Ferner kann »trafficLight« nur einen vordefinierten und keinen eigenen, benutzerdefinierten Typ haben. Es können daher der Variablen »trafficLight« beliebige, also auch nicht mehr sinnvoll interpretierbare Werte vom Typ int zugewiesen werden, z.B. 17. Ist man nicht bereit, diese Nachteile bezüglich Lesbarkeit und Ausführungssicherheit zu akzeptieren, dann muss eine eigene Klasse Ampel geschaffen werden (c). Generell gilt: Für alle nicht privaten Klassen ist – obwohl aufwendiger – eine Lösung nach Schema (c) vorzuziehen, da sie zu robusterem Code führt, die Entwurfsentscheidung, wie »trafficLight« realisiert ist, kapselt und über die Schnittstelle nur sinnvoll interpretierbare Objektzustände erlaubt …

ABBILDUNG 3 Substituieren von Aufzählungstypen. Beispiel für eine Regelung der Kategorie (1)

das (syntaktische) Auflösen der Dangling-Else-Mehrdeutigkeit (siehe Abbildung 2). Insgesamt betrachtet ist diese Kategorie in Java aber erfreulich klein.

Zur zweiten Kategorie (2) gehören alle Probleme, die nicht oder nur sehr umständlich über die Sprachdefinition zu lösen sind, wie beispielsweise das Ändern von Code (insbesondere fremden Codes), das generelle Layout des Codes, die Bezeichnerwahl sowie die Dokumentation und Kommentierung.

Anzumerken ist noch, dass die beiden Kategorien nicht vollständig disjunkt sind, d.h. manches ist nicht eindeutig einer Kategorie zuzuordnen, beispielsweise das Layout des Codes. Als Faustregel, um die Wichtigkeit einzelner Regelungen abzuschätzen, ist diese Unterscheidung dennoch nützlich, insbesondere bei der Prüfung der Richtlinien.

4.1 Auszüge aus den Richtlinien

Dieser Abschnitt zeigt drei Beispiele (Abbildungen 2-4), wie Richtlinien dokumentiert werden [Berner et. al. 96]. Die ersten beiden Beispiele fallen schwerpunktmässig unter die Kategorie (1). Danach folgt ein Beispiel für eine Regelungen der Kategorie (2).

Probleme der Kategorie (1) lassen sich meist einfach auf eine recht formale Art regeln, indem man die Verwendung bestimmter Sprachkonstrukte verbietet, einschränkt oder erschwert. Beispielsweise, indem man eine explizite Klammerung für jede if-Anweisung vorschreibt (Abbildung 2) oder indem man geeignete Substitute für fehlende bzw. nicht vorhandene Sprachelemente angibt (Abbildung 3). Regelungen, die Probleme der Kategorie (1) behandeln, dienen hauptsächlich dazu, Sprachschwächen zu entschärfen und Flüchtigkeitsfehler zu verhindern. Solche Regelungen sind typisch syntaktischer Natur und lassen sich weitgehend formal beschreiben. Die Abbildungen 2 und 3 zeigen Beispiele für Regelungen dieser Art. Abgesehen davon, dass der Entwickler ggf. ein bestimmtes Problemlösungsschema wiedererkennen kann und ihm dies das schnelle Erfassen bestimmter Code-Fragmente erleichtert, ist ihr genereller Nutzen für das bessere Verständnis des Programmcodes eher gering.

Abbildung 4 zeigt eine Regelung der Kategorie (2). Diese Regelungen betreffen primär Bereiche, die nur sehr restriktiv oder überhaupt nicht über die Sprachdefinition zu regeln sind und zielen primär auf Aspekte wie Programmierstil sowie einfaches Verstehen und gute Lesbarkeit des Codes ab. Regelungen dieser Kategorie sollen gezielt dazu beitragen, Inhalt und Bedeutung des Codes möglichst klar und verständlich zu halten. Wenn man zugesteht, dass insbesondere bei Wartung von Software das Lesen und Verstehen von Code eine zentrale Aktivität ist, dann ist ihr Nutzen offensichtlich.

Die meisten Probleme der Kategorie (2) lassen sich nicht in formale Regeln fassen, sondern müssen inhaltlich durch Texte beschrieben werden. Dabei kommt verständlichen Beispielen eine zentrale Rolle zu. Die Anwendung und Überprüfung dieser Regelungen gestaltet sich aufwendiger und schwieriger als bei Regelungen der Kategorie (1), da ein grundlegendes, inhaltliches Verständnis des Codes Voraussetzung dafür ist, die Regelungen anzuwenden oder zu überprüfen. So ist es beispielsweise wesentlich einfacher und eindeutiger festzustellen, dass die Klammerung einer if-Anweisung fehlt (Abbildung 1), als die Verständlichkeit eines Bezeichners konstruktiv zu beurteilen.

2.4 Klassen

Klassenkopf

Neben Klassenbezeichnung, Klassenkommentar und ggf. Schnittstellen wird im Klassenkopf auch die Attributierung der Klasse geregelt, d.h. es wird festgelegt, ob die Klasse abstrakt ist (Schlüsselwort *abstract*), ob es Unterklassen geben darf (Schlüsselwort *final*) und wie die Sichtbarkeit des Klassenbezeichners ist (Schlüsselwort *public*). Hierbei sollten die folgenden Regeln beachtet werden:

Abstrakte Klassen

Eine Klasse sollte nur dann *abstract* attributiert werden, wenn sie »teilweise abstrakt« ist, d.h. wenn sie eine bestimmte Funktionalität implementiert, die sie mit ihren (potentiellen) Unterklassen gemeinsam hat. Wenn Unsicherheit oder Unklarheit darüber besteht, in welcher Form die Funktionalität zu implementieren ist oder wenn es darum geht, ein bestimmtes Protokoll oder ein bestimmtes Schnittstellenformat einzuhalten, dann sollten Interfaces und nicht abstrakte Klassen verwendet werden. Interfaces sind wesentlich flexibler als abstrakte Klassen. Sie unterstützen Mehrfachvererbung und können u.a. auch dazu benutzt werden, ansonsten unzusammenhängenden Klassen ähnliche Funktionalität zu geben.

"final" Klassen

Eine Klasse sollte nur dann *final* attributiert sein, wenn sie Unterklasse oder Implementierung einer Schnittstelle ist, welche bereits alle Methoden definiert, die nicht implementierungsspezifisch für die jeweilige Klasse sind. Wenn eine Klasse »nur« *final* attributiert wird, dann kann von dieser Klasse keine Unterklasse mehr gebildet werden. Existiert zu dieser Klasse eine Basisklasse, die nicht *final* attributiert ist, aber gleiche Funktionalität hat, dann besteht zumindest die Möglichkeit, von dieser Klassen eine Unterklasse zu bilden [siehe auch Lea 1996].

ABBILDUNG 4 Attributierung von Klassen. Beispiel für eine Regelung der Kategorie (2) aus dem Kapitel »Programmierhinweise für Java«.

5 Anwendung und Änderung

Nur wenn Richtlinien »gelebt« werden, also nicht ausschliesslich auf dem Papier existieren, sondern auch in den Köpfen der Entwickler, haben sie einen

Nutzen. »Gelebt werden« heisst in diesem Zusammenhang: Die Richtlinien müssen den Entwicklern bekannt gemacht und von diesen angenommen werden. Es muss regelmässig geprüft werden, ob und inwiefern die Richtlinien eingehalten werden. Ebenso müssen die Richtlinien beständig an geänderte Gegebenheiten angepasst werden, damit sie auf Dauer anwendbar bleiben.

Geprüft wird die Einhaltung der Richtlinien zweckmässigerweise im Rahmen von Code-Inspektionen. Hierzu macht man die Richtlinien zum Bestandteil der Prüfkriterien. Hauptsächlich sollte die inhaltliche Einhaltung geprüft werden (siehe Probleme der Kategorie (2)), also Aspekte wie Programmierstil, Bezeichnerwahl, Kommentierung, etc. Verstösse gegen formale Regelungen wie Einrückung und Layout (siehe Probleme der Kategorie (1)) fallen bei der Inspektion eher in die Klasse der nebensächlichen Befunde. Dies ist überwiegend dadurch motiviert, dass sich das Einhalten formaler Aspekte weitaus einfacher einhalten und überprüfen lässt, als beispielsweise eine geeignete Namensgebung für Bezeichner. Ausserdem konzentrieren bzw. beschränken sich insbesondere weniger gut vorbereitete Gutachter oft auf die Einhaltung formaler Aspekte.

Entwicklungsrichtlinien sind keine »zementierten«, auf alle Zeit hin zu beachtenden Regelwerke. Damit Richtlinien wirklich gelebt werden, ist es unerlässlich, sie stetig und umsichtig zu aktualisieren, d.h. überholte Regelungen müssen entfernt und ggf. neue hinzugefügt werden. Neben einem angebrachten Mass an Vereinheitlichung besteht ein Hauptnutzen von Entwicklungsrichtlinien ja darin, gemachte Erfahrungen sowie die daraus gezogenen Lehren aufzubereiten und zu so dokumentieren, dass möglichst viele davon profitieren.

Daraus resultiert auch die Verpflichtung, Verbesserungsvorschläge derjenigen Personen, die auf der Grundlage der Richtlinien arbeiten bzw. entwickeln, schnell zu berücksichtigen und in eine überarbeitete Versionen der Richtlinien einfliessen zu lassen. Hierzu bietet es sich an, Verbesserungsvorschläge konsequent zu sammeln und in regelmässigen Abständen in die Richtlinien einzuarbeiten. Eine neue Version der Richtlinien wird dann in einem Review geprüft und anschliessend freigegeben.

Ω

6 Literatur

Referenzen

[Berner et al. 96]
Berner, S., M. Arnold, M. Glinz, S. Joos (1996): *Entwicklungsrichtlinien für Java-Software – Version 1.4.2*. Institut für Informatik der Universität Zürich, 1996. [http://www.ifi.unizh.ch/groups/req/publications/selected_publications.html]

[Lea96]
Lea, D. (1996): *Draft Java Coding Standard*. [http://g.oswego.edu/dl/html/javaCodingStd.html]

[Schneider94]
Schneider, K. (1994): *Styleguide für Smalltalk-Entwicklungen – Version 3 und 4*. Abteilung Software Engineering, Institut für Informatik der Universität Stuttgart, 1992 und 1994.

Weitere Literatur

[Lorenz93]
Lorenz, M. (1993): *Object-Oriented Software Development – Practical Guide*. Prentice Hall, Englewood Cliffs, 1993.

[Meyer88]
Meyer, B. (1988): *Object-Oriented Software Construction*. Prentice Hall International, London, 1988.